IDA PAPER P-2109

Ada LEXICAL ANALYZER GENERATOR USER'S GUIDE

Reginald N. Meeson

January 1989

Prepared for
STARS Joint Program Office

IDA
INSTITUTE FOR DEFENSE ANALYSES
1801 N. Beauregard Street, Alexandria, Virginia 22311-1772

IDA Log No. HQ 88-033376

DEFINITIONS

IDA publishes the following documents to report the results of its work.

Reports

Reports are the most authoritative and most carefully considered products IDA publishes. They normally embody results of major projects which (a) have a direct bearing on decisions affecting major programs, (b) address issues of significant concern to the Executive Branch, the Congress and/or the public, or (c) address issues that have significant economic implications. IDA Reports are reviewed by outside panels of experts to ensure their high quality and relevance to the problems studied, and they are released by the President of IDA.

Group Reports

Group Reports record the findings and results of IDA established working groups and panels composed of senior individuals addressing major issues which otherwise would be the subject of an IDA Report. IDA Group Reports are reviewed by the senior individuals responsible for the project and others as selected by IDA to ensure their high quality and relevance to the problems studied, and are released by the President of IDA.

Papers

Papers, also authoritative and carefully considered products of IDA, address studies that are narrower in scope than those covered in Reports. IDA Papers are reviewed to ensure that they meet the high standards expected of refereed papers in professional journals or formal Agency reports.

Documents

IDA Documents are used for the convenience of the sponsors or the analysts (a) to record substantive work done in quick reaction studies, (b) to record the proceedings of conferences and meetings, (c) to make available preliminary and tentative results of analyses, (d) to record data developed in the course of an investigation, or (e) to forward information that is essentially unanalyzed and unevaluated. The review of IDA Documents is suited to their content and intended use.

The work reported in this document was conducted under contract MDA 903 84 C 0031 for the Department of Defense. The publication of this IDA document does not indicate endorsement by the Department of Defense, nor should the contents be construed as reflecting the official position of that Agency.

This Paper has been reviewed by IDA to assure that it meets high standards of thoroughness, objectivity, and appropriate analytical methodology and that the results, conclusions and recommendations are properly supported by the material presented.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*

2. REPORT DATE
January 1989

3. REPORT TYPE AND DATES COVERED
Final

4. TITLE AND SUBTITLE
Ada Lexical Analyzer Generator User's Guide

5. FUNDING NUMBERS
MDA 903 84 C 0031
A-134

6. AUTHOR(S)
Reginald N. Meeson

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Institute for Defense Analyses
1801 N. Beauregard St.
Alexandria, VA 22311-1772

8. PERFORMING ORGANIZATION REPORT NUMBER
IDA Paper P-2109

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
STARS Joint Program Office
1400 Wilson Blvd.
Arlington, VA 22209-2308

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release, unlimited distribution; 30 August 1990.

12b. DISTRIBUTION CODE
2A

13. ABSTRACT *(Maximum 200 words)*

IDA Paper P-2109, Ada Lexical Analyzer Generator User's Guide, documents how to use the Ada Lexical Analyzer Generator program which will create a lexical analyzer or "next-token" procedure for use in a compiler, pretty printer, or other language processing programs. Lexical analyzers are produced from specifications of the patterns they must recognize. The notation for specifying patterns is essentially the same as that used in the Ada Language Reference Manual. The generator produces an Ada package that includes code to match the specified lexical patterns and returns the symbols it recognizes. Familiarity with compiler terminology and techniques is assumed in the technical sections of this document. Automated generation of lexical analyzers is illustrated by developing a complete example.

14. SUBJECT TERMS
Ada Programming Language; Software Engineering; Lexical Analysis; Lexical Patterns.

15. NUMBER OF PAGES
50

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18
298-102

IDA PAPER P-2109

Ada LEXICAL ANALYZER GENERATOR USER'S GUIDE

Reginald N. Meeson

January 1989

INSTITUTE FOR DEFENSE ANALYSES

Contract MDA 903 84 C 0031

DARPA Assignment A-134

PREFACE

The purpose of IDA Paper P-2109, *Ada Lexical Analyzer Generator User's Guide*, is to document the use of the Ada Lexical Analyzer Generator, which was developed as part of IDA's prototype software development work for the Software Technology for Adaptable and Reliable Software (STARS) program under Task Order T-D5-429. This paper is directed toward potential users of the generator program. Automated generation of lexical analyzers is illustrated by developing a complete example.

An earlier draft of this document was reviewed within the Computer and Software Engineering Division (CSED) by B. Brykczynski, W. Easton, R. Knapper, J. Sensiba, L. Veren, R. Waychoff, and R. Winner (April 1988).

CONTENTS

1. INTRODUCTION

This document is a user's guide for the Ada Lexical Analyzer Generator. The generator was developed as part of IDA's prototype software development work for STARS (Software Technology for Adaptable and Reliable Systems). This report was written in partial fulfillment of Section 4(d) of Task Order T-D5-429.

The Ada Lexical Analyzer Generator is a program that will create a lexical analyzer or "next-token" procedure for use in a compiler, pretty printer, or other language processing program. Lexical analyzers are produced from specifications of the patterns they must recognize. The notation for specifying patterns is essentially the same as that used in the Ada language reference manual [1]. The generator produces an Ada package that includes code to match the specified lexical patterns and return the symbols it recognizes. Familiarity with Ada programming techniques is assumed in the technical sections of this report.

1.1 Scope

This report describes the notation used for specifying lexical patterns and "walks through" the process of creating a lexicon and generating a simple lexical analyzer.

1.2 Background

Lexical analysis is the first stage of processing in a compiler or other language processing program, and is where basic language elements such as identifiers, numbers, and special symbols are separated from the sequence of characters submitted as input. Lexical analysis does not include recognizing higher levels of source language structure such as expressions or statements. This processing is performed by the next compiler stage, the parser. Separating the lexical analysis stage from the parsing stage greatly simplifies the parser's task. Lexical analyzers also simplify other language processing tools that do not need full-scale parsers for their entire input language; for example, pretty printers. In fact, lexical analysis techniques can simplify many other applications that process complex input data.

For more information on compiler organization and implementation techniques, readers may wish to consult a standard text on compiler development. (See, for example, the "dragon" book [2].)

A lexical analyzer generator produces lexical analyzers automatically from specifications of the lexical components of the input language. This is easier and more reliable than coding lexical analyzers manually. One commercial lexical analyzer generator on the market is the Unix™-based program "lex" [3]. The Ada lexical analyzer generator differs from lex in at least three significant ways:

- The notation for describing lexical patterns is much easier to read and understand
- The generator produces directly executable code (lex-generated analyzers are table driven)
- The generator produces Ada code

2. LEXICON DEFINITION

This section presents the basic rules for creating specifications for lexical analyzers. This includes defining the patterns to be matched, and the actions to be taken when patterns are recognized.

Different type faces are used below to distinguish rules for writing lexical analyzer specifications from examples of pattern definitions. This device is necessary because, by design, the notation for patterns is almost identical to that for syntax rules.

- Rules for writing specifications are set in -- plain font
- Pattern definitions and sample code are set in -- `typewriter font`

2.1 Lexical Pattern Notation

Lexical patterns are specified using a simple variant of Backus-Naur Form (BNF). Definitions in this language follow the rule

```
pattern_definition ::=
      pattern_name "::=" regular_expression ";"
```

Pattern names are represented by Ada identifiers. The symbol "::=" is the pattern definition operator. Regular expressions are made up of literal symbols and pattern names using the combining forms described below. Pattern definitions are always terminated with a semicolon.

Literal symbols are represented by Ada character and string literals and by reserved identifiers. For example,

```
Semicolon  ::=  ';' ;
Apostrophe  ::=  ''' ;
Assignment_Symbol  ::=  ":=" ;
```

Patterns can be concatenated by writing them consecutively, without an operator symbol, as in

```
Character_Literal  ::=  Apostrophe Graphic_Character Apostrophe ;
```

Literal string values are equivalent to the concatenation of the corresponding literal characters. For example, the string ":=" is the same as the concatenation of the two characters ':' and '='.

Character ranges can be specified using Ada's double-dot notation. For example,

```
Digit  ::=  '0' .. '9' ;
Lower_Case_Letter  ::=  'a' .. 'z' ;
Upper_Case_Letter  ::=  'A' .. 'Z' ;
```

A vertical bar is used to separate alternative patterns, as in

```
Letter  ::=  Lower_Case_Letter | Upper_Case_Letter ;
Letter_or_Digit  ::=  Letter | Digit ;
```

Square brackets are used to enclose optional patterns. For example, numbers with optional fraction and exponent parts can be specified by

```
Decimal_Literal  ::=  Integer ['.' Integer] [Exponent] ;
```

Braces are used to enclose a repeated pattern, as in the following expression for identifiers. The enclosed pattern may be matched zero or more times. Examples are

```
Identifier  ::=  Letter { ['_'] Letter_or_Digit } ;
Integer  ::=  Digit { ['_'] Digit } ;
```

Options and repetitions are exercised whenever possible so that the longest possible pattern is always matched.

Precedence of operations. Of the three infix pattern construction operations, range construction has the highest precedence, so ranges are always constructed first. Only character ranges can be constructed. Concatenation is next. Any literal, range, or named pattern can be concatenated to another. Alternation has the lowest precedence and is always performed last.

The notation does not include parentheses to override these precedence rules. The effect can be achieved, however, by defining additional patterns. The pattern for identifiers is an example of this. The sub-patterns Letter and Letter_or_Digit force the alternations to be formed before the concatenations.

Regular form. To allow simple, efficient code to be generated for lexical analyzers, the input pattern definitions must have a simple structure. Specifically, they must form a regular grammar so that code for an equivalent finite-state machine can be generated. The pattern construction operations described above allow the definition of arbitrary regular patterns. The lexical analyzer generator does not support recursive pattern definitions.

Predefined patterns. The patterns END_OF_INPUT, END_OF_LINE, and UNRECOGNIZED are automatically defined and handled by the generated code.

2.2 Declarations and Actions

In addition to the specification of lexical patterns, the lexical analyzer generator requires definitions of the actions to be taken when a pattern is recognized. These actions may further require type, variable, and procedure declarations to be included in the package that is created. Lexical analyzer specifications, therefore, follow the rule:

```
lexical_analyzer_specification ::=
        lexicon token_stream_name is
                [ declarative_part ]
        patterns
                { pattern_definition }
        actions
                { action_alternative }
        end [token_stream_name] ";"
```

"Lexicon" is a reserved word. The token stream name is the name of the token stream package generated by the lexical analyzer. The declarative part allows the declaration of any supporting constants, types, variables, functions, or procedures. These declarations are copied into the generated package body.

"Patterns" is a reserved word. Pattern definitions have the form described above.

"Actions" is a reserved word. Action alternatives follow the same rule as Ada case statement alternatives; that is,

```
action_alternative ::=
        when choice {"|" choice} "=>" sequence_of_statements
```

Action choices can be any defined pattern name or "others" for the last action alternative. The generator turns the action alternatives into a case statement with the name of the recognized pattern as the selector.

There are two principle actions a lexical analyzer performs, returning a token value and skipping over uninteresting input. To return a token to the calling program, the action statements must assign a value to the output parameter NEXT (see Section 3.1) and end with a "return" statement. For example,

```
when  Identifier  =>
   NEXT := MAKE_TOKEN( IDENT, CURRENT_SYMBOL, CUR_LINE_NUM );
   return;
```

To skip over a recognized pattern (for example, white space or comments), specify "null" as the action, with no return. For example,

```
when  White_Space  =>  null;
```

The parameterless function CURRENT_SYMBOL returns the recognized string. CUR_LINE_NUM is an integer variable that holds the current line number.

3. PROGRAM INTERFACES

This section describes the input and output interfaces for generated lexical analyzers and the options available for incorporating generated code into application programs.

3.1 Input and Output Streams

The input character stream for the lexical analyzer is represented by a procedure that produces consecutive characters on each call. The specification for this procedure is

```
procedure  GET_CHARACTER( EOS: out BOOLEAN;
                          NEXT: out CHARACTER;
                          MORE: in BOOLEAN := TRUE );
```

This mechanism allows input text to be produced from a file or from other sources within a program.

The output stream produced by the lexical analyzer generator is a sequence of tokens. The specification for the token stream package generated is

```
package  TOKEN_STREAM_NAME  is

   procedure  ADVANCE( EOS: out BOOLEAN;
                       NEXT: out TOKEN;
                       MORE: in BOOLEAN := TRUE );

end  TOKEN_STREAM_NAME;
```

The package name is taken from the lexicon specification. The procedure ADVANCE reads input by invoking the GET_CHARACTER procedure. It returns an end-of-stream flag, EOS, which is TRUE when the end of the input is reached. When EOS is FALSE, NEXT contains the next token value. TOKEN is a user-defined type. The optional parameter MORE may be set to FALSE to indicate that no more tokens will be drawn from the stream.

3.2 Packaging Options

There are three methods for combining generated stream packages with the remainder of an application program:

- Copying the generated text into the program source file
- Making the generated package body a separate compilation unit
- Creating a generic package

Copying generated text is the least flexible method. If any of the lexical patterns are changed, the old text must be extracted and replaced by the new using a text editor. Creating a generic package requires passing the GET_CHARACTER procedure and TOKEN type, and possibly other information, as instantiation parameters. Making the package body a separate compilation unit is the simplest method. Generics and separate compilation are supported by the generator by allowing either a generic formal part or a "separate" declaration to precede a lexical analyzer specification. A complete description of the rule for specifications is

```
lexical_analyzer_specification ::=
        [ context_clause ]
        [ generic_formal_part | separate "(" parent_name ")" ]
        lexicon token_stream_name is
                [ declarative_part ]
        patterns
                { pattern_definition }
        actions
                { action_alternative }
        end [token_stream_name] ";"
```

For generic lexical analyzers, a complete package definition (specification and body) with the specified generic parameters is generated. The GET_CHARACTER procedure and TOKEN type must be included in the list of generic parameters. For non-generic analyzers, only the package body is generated. If a "separate" clause is supplied in the lexicon specification, it is reproduced in the generated code. The parent unit must include the package specification and an "is separate" declaration for the package body.

3.3 Generator Files

The generator reads lexical analyzer specifications from the STANDARD_INPUT file and writes its output to the STANDARD_OUTPUT file. These input and output interfaces may be redirected to appropriate files using available operating system commands. Error messages are written to a file called STANDARD_ERROR. The generator reads one additional file called TABLE, which contains its translation tables.

4. SAMPLE SPECIFICATION

In this section a complete specification for a simple lexical analyzer is developed. This analyzer will be required to recognize and return the components of arithmetic expressions, skipping white space and comments. Specifically, the components to be recognized are

- Identifiers for variables and functions
- Integer and real decimal numbers
- Operator symbols ("+", "-", "*", and "/")
- Left and right parentheses

4.1 Patterns

Most of the pieces for the required patterns were introduced in Section 2, so it should be fairly easy to create the "patterns" section of the specification. For identifiers and numbers we have:

```
Identifier  ::=  Letter { ['_'] Letter_or_Digit } ;

Letter  ::=  'A'..'Z' | 'a'..'z' ;

Digit  ::=  '0'..'9' ;

Letter_or_Digit  ::=  Letter | Digit ;

Integer   ::=  Digit { ['_'] Digit } ;

Decimal_Literal  ::=  Integer ['.' Integer] [Exponent] ;
```

The remaining patterns needed are:

```
Exponent  ::=  'E' ['+'|'-'] Integer ;

Operator_Symbol  ::=  '+' | '-' | '*' | '/' ;

Left_Parenthesis  ::=  '(' ;

Right_Parenthesis  ::=  ')' ;

Comment  ::=  "--" { Graphic_Character | ASCII.HT } ;

Graphic_Character  ::=  ' '..'~' ;

White_Space  ::=  Separator { Separator } ;
```

```
Separator  ::=  ' ' | ASCII.HT ;
```

Any other text that appears in the input, such as other special symbols, will be matched by the UNRECOGNIZED pattern.

4.2 Actions

Actions must be specified for each of the patterns to be recognized. To simplify this discussion, assume there exists a function called "MAKE_TOKEN" that creates a token value from the information collected by the lexical analyzer. The requirements for this function will be clear from its use. Assume also that there exists an enumerated type that identifies the type of token returned. For this example we will use

```
type  TOKEN_TYPE  is
         (IDENT, LF_PAREN, NOT_MINE, NUMBER, OPERATOR, RT_PAREN);
```

The actions that return tokens can then be specified by the following "when" clauses:

```
when  Identifier  =>
   NEXT := MAKE_TOKEN( IDENT, CURRENT_SYMBOL, CUR_LINE_NUM );
   return;

when  Decimal_Literal  =>
   NEXT := MAKE_TOKEN( NUMBER, CURRENT_SYMBOL, CUR_LINE_NUM );
   return;

when  Operator_Symbol  =>
   NEXT := MAKE_TOKEN( OPERATOR, CURRENT_SYMBOL, CUR_LINE_NUM );
   return;

when  Left_Parenthesis  =>
   NEXT := MAKE_TOKEN( LF_PAREN, CURRENT_SYMBOL, CUR_LINE_NUM );
   return;

when  Right_Parenthesis  =>
   NEXT := MAKE_TOKEN( RT_PAREN, CURRENT_SYMBOL, CUR_LINE_NUM );
   return;
```

For comments and white space the action is to skip over the input text and search for the next pattern. This is achieved by the clause

```
when  Comment | White_Space  =>  null ;
```

Actions must also be specified for input that does not match any pattern. One option is to skip over such input. This prevents the calling procedure from handling input errors intelligently, however, because it never sees the errors. One solution is to return an "unrecognized" token and let the calling procedure deal with the problem.

```
when others  =>
   NEXT := MAKE_TOKEN( NOT_MINE, CURRENT_SYMBOL, CUR_LINE_NUM );
   return;
```

This completes the list of required actions.

4.3 Packaging

For this example a lexical analyzer with a separately compiled package body will be generated. This is accomplished by the following outline of the complete specification.

```
separate ( SAMPLE_TEST_PROGRAM )

lexicon  SAMPLE_TOKEN_STREAM  is

patterns
   -- include all the pattern definitions here

actions
   -- include all the action "when" clauses here

end  SAMPLE_TOKEN_STREAM ;
```

The lexical analyzer generator will produce a separately compilable package body for a package named SAMPLE_STREAM. As described above, the calling procedure must contain the specification for this package and the clause

```
package body  SAMPLE_TOKEN_STREAM  is separate;
```

The typc TOKEN, the procedure GET_CHARACTER, and the function MAKE_TOKEN, which was used to construct token values, must also be defined and must be visible within the calling procedure or program.

Appendix A presents a complete listing of the specification for this sample lexical analyzer. A pretty-printed listing of code produced by the generator is presented in Appendix B. Appendix C presents a test program that prints the tokens it receives from the analyzer. Appendix D shows sample input data and the output produced by this program.

5. ADDITIONAL FEATURES

This section discusses some additional features of the lexical analyzer generator that are not illustrated in the examples above.

5.1 Additional Alternatives

Alternatives for a pattern may be specified by creating multiple definitions for the same pattern name. For example, if relational operations were required in addition to the arithmetic operations in the lexical analyzer we created above, the following definition could be added to the existing list of patterns.

```
Operator_Symbol  ::=  "/=" | '<' | "<=" | '=' | '>' | ">=" ;
```

5.2 Look-Ahead and Ambiguity

Two different patterns may start with the same character or sequence of characters. This requires lexical analyzers to "look" ahead into the input to determine which pattern to match. This look-ahead processing can usually be handled completely automatically.

Patterns may also be ambiguous. That is, a given sequence of characters may match two different patterns at the same time. Normal processing attempts to match the longer pattern first and accept it if it matches. If the longer pattern fails to match, the analyzer will fall back and match the shorter pattern.

To match the shorter of two ambiguous patterns, a special look-ahead operator is provided. The classic example of this situation is the Fortran "DO" statement. The following Fortran statements illustrate the problem:

`DO 10 I = 1,10` and `DO 10 I = 1.10`

The first is the start of a loop structure, for which the keyword "DO" must be matched. The second is an assignment statement, for which the identifier "DO10I" must be matched. Without special attention, the analyzer would match identifier "DO10I" in both cases. The pattern required to recognize the keyword "DO" is

```
Keyword_DO  ::=  "DO" # Label Identifier '=' Index_Expr ',' ;
```

The sharp symbol (#, not in quotes) separates this pattern into two parts. If the entire pattern is matched the analyzer falls back to the # and returns the first part of the pattern as the result. The string to the right is preserved as input to be scanned for the next symbol, which in this example is the loop label. If the pattern fails to match, the lexical analyzer falls back to the # and attempts to match the alternative pattern, which in this example is an identifier.

6. REFERENCES

[1] *Ada Programming Language,* ANSI/MIL-STD-1815A, January 1983.

[2] Aho, A., R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools,* Addison-Wesley, 1985.

[3] Lesk, M., *Lex -- A Lexical Analyzer Generator,* Computing Science Technical Report 39, AT&T Bell Laboratories, Murray Hill, NJ, 1975.

APPENDIX A - SAMPLE LEXICAL ANALYZER SPECIFICATION

This appendix contains a listing of the sample lexical analyzer specification discussed in Section 4. This specification contains pattern definitions and action statements necessary to construct a lexical analyzer that will recognize components of simple arithmetic expressions.

```
separate ( SAMPLE_TEST_PROGRAM )

lexicon  SAMPLE_TOKEN_STREAM  is

patterns

   Comment  ::=  "--" { Graphic_Character | ASCII.HT } ;

   Decimal_Literal  ::=  Integer ['.' Integer] [ Exponent ] ;

   Digit  ::=  '0' .. '9' ;

   Exponent  ::=  'E' ['+' | '-'] Integer ;

   Graphic_Character  ::=  ' ' .. '~' ;

   Identifier  ::=  Letter { ['_'] Letter_or_Digit } ;

   Integer  ::=  Digit { ['_'] Digit } ;

   Left_Parenthesis  ::=  '(' ;

   Letter  ::=  'A' .. 'Z' | 'a' .. 'z' ;

   Letter_or_Digit  ::=  Letter | Digit ;

   Operator_Symbol  ::=  '+' | '-' | '*' | '/' ;

   Right_Parenthesis  ::=  ')' ;

   White_Space  ::=  ' ' { ' ' };

actions

   when  Comment | White_Space  =>  null;

   when  Decimal_Literal  =>
      NEXT := MAKE_TOKEN( NUMBER, CURRENT_SYMBOL, CUR_LINE_NUM );
      return;

   when  Identifier  =>
      NEXT := MAKE_TOKEN( IDENT, CURRENT_SYMBOL, CUR_LINE_NUM );
      return;

   when  Left_Parenthesis  =>
      NEXT := MAKE_TOKEN( LF_PAREN, CURRENT_SYMBOL, CUR_LINE_NUM );
      return;

   when  Operator_Symbol  =>
```

```
        NEXT := MAKE_TOKEN( OPERATOR, CURRENT_SYMBOL, CUR_LINE_NUM );
        return;

     when  Right_Parenthesis  =>
        NEXT := MAKE_TOKEN( RT_PAREN, CURRENT_SYMBOL, CUR_LINE_NUM );
        return;

     when others  =>
        NEXT := MAKE_TOKEN( NOT_MINE, CURRENT_SYMBOL, CUR_LINE_NUM );
        return;

  end  SAMPLE_TOKEN_STREAM;
```

APPENDIX B - SAMPLE GENERATED LEXICAL ANALYZER CODE

This appendix contains a pretty-printed listing of code generated for the sample lexical analyzer discussed in Section 4. This package body includes pattern-matching code for all specified lexical patterns and actions to be taken when patterns are recognized. This code was generated automatically by the lexical analyzer generator from the specification given in Appendix A.

```
separate ( SAMPLE_TEST_PROGRAM )

package body SAMPLE_TOKEN_STREAM is

  BUFFER_SIZE: constant := 100;
  subtype BUFFER_INDEX is INTEGER range 1..BUFFER_SIZE;

  type PATTERN_ID is
    (Comment,Decimal_Literal,Digit,Exponent,Graphic_Character,
     Identifier,Integer,Left_Parenthesis,Letter,Letter_or_Digit,
     Operator_Symbol ,Right_Parenthesis,White_Space,
     END_OF_INPUT, END_OF_LINE, UNRECOGNIZED);

  CUR_LINE_NUM: NATURAL := 0;
  CUR_PATTERN: PATTERN_ID := END_OF_LINE;
  START_OF_LINE: BOOLEAN;
  CHAR_BUFFER: STRING(BUFFER_INDEX);
  CUR_CHAR_NDX: BUFFER_INDEX;
  TOP_CHAR_NDX: BUFFER_INDEX;

  procedure SCAN_PATTERN;  -- forward

  function CURRENT_SYMBOL return STRING is
  begin
    return CHAR_BUFFER(1..(CUR_CHAR_NDX-1));
  end;

  procedure ADVANCE(EOS: out BOOLEAN;
    NEXT: out TOKEN;
    MORE: in BOOLEAN := TRUE) is
  begin
    EOS := FALSE;
    loop
      SCAN_PATTERN;
      case CUR_PATTERN is
        when END_OF_INPUT =>
          EOS := TRUE;
          return;
        when END_OF_LINE => null;
        when Comment | White_Space => null;
        when Decimal_Literal =>
          NEXT := MAKE_TOKEN( NUMBER, CURRENT_SYMBOL, CUR_LINE_NUM);
          return;
        when Identifier =>
          NEXT := MAKE_TOKEN( IDENT, CURRENT_SYMBOL, CUR_LINE_NUM);
          return;
        when Left_Parenthesis =>
          NEXT := MAKE_TOKEN( LF_PAREN, CURRENT_SYMBOL, CUR_LINE_NUM);
          return;
```

```
      when Operator_Symbol =>
        NEXT := MAKE_TOKEN( OPERATOR, CURRENT_SYMBOL, CUR_LINE_NUM);
        return;
      when Right_Parenthesis =>
        NEXT := MAKE_TOKEN( RT_PAREN, CURRENT_SYMBOL, CUR_LINE_NUM);
        return;
      when others =>
        NEXT := MAKE_TOKEN( NOT_MINE, CURRENT_SYMBOL, CUR_LINE_NUM);
        return;
    end case;
  end loop;
end ADVANCE;


procedure SCAN_PATTERN is

  CURRENT_CHAR: CHARACTER;
  END_OF_INPUT_STREAM: BOOLEAN;
  LOOK_AHEAD_FAILED: BOOLEAN := FALSE;
  FALL_BACK_NDX: BUFFER_INDEX := 1;
  LOOK_AHEAD_NDX: BUFFER_INDEX;

  procedure CHAR_ADVANCE is
  begin
    CUR_CHAR_NDX := CUR_CHAR_NDX+1;
    FALL_BACK_NDX := CUR_CHAR_NDX;
    if CUR_CHAR_NDX <= TOP_CHAR_NDX then
      CURRENT_CHAR := CHAR_BUFFER(CUR_CHAR_NDX);
    else
      GET_CHARACTER(END_OF_INPUT_STREAM,CURRENT_CHAR);
      if END_OF_INPUT_STREAM then
        CURRENT_CHAR := ASCII.etx;
      end if;
      CHAR_BUFFER(CUR_CHAR_NDX) := CURRENT_CHAR;
      TOP_CHAR_NDX := CUR_CHAR_NDX;
    end if;
  end;

  procedure LOOK_AHEAD is
  begin
    CUR_CHAR_NDX := CUR_CHAR_NDX+1;
    if CUR_CHAR_NDX <= TOP_CHAR_NDX then
      CURRENT_CHAR := CHAR_BUFFER(CUR_CHAR_NDX);
    else
      GET_CHARACTER(END_OF_INPUT_STREAM,CURRENT_CHAR);
      if END_OF_INPUT_STREAM then
        CURRENT_CHAR := ASCII.etx;
      end if;
      CHAR_BUFFER(CUR_CHAR_NDX) := CURRENT_CHAR;
```

```
      TOP_CHAR_NDX := CUR_CHAR_NDX;
    end if;
  end;

begin
  START_OF_LINE := CUR_PATTERN = END_OF_LINE;
  if START_OF_LINE then
    CUR_LINE_NUM := CUR_LINE_NUM+1;
    TOP_CHAR_NDX := 1;
    GET_CHARACTER(END_OF_INPUT_STREAM,CHAR_BUFFER(1));
    if END_OF_INPUT_STREAM then
      CHAR_BUFFER(1) := ASCII.etx;
    end if;
  else
    TOP_CHAR_NDX := TOP_CHAR_NDX-CUR_CHAR_NDX+1;
    for N in 1..TOP_CHAR_NDX loop
      CHAR_BUFFER(N) := CHAR_BUFFER(N+CUR_CHAR_NDX-1);
    end loop;
  end if;
  CUR_CHAR_NDX := 1;
  CURRENT_CHAR := CHAR_BUFFER(1);
  case CURRENT_CHAR is
    when ASCII.etx =>
      CUR_PATTERN := END_OF_INPUT;
    when ASCII.lf..ASCII.cr =>
      CUR_PATTERN := END_OF_LINE;
    when ')' =>
      CHAR_ADVANCE;
      CUR_PATTERN := Right_Parenthesis;
    when '*'..'+' | '/' =>
      CHAR_ADVANCE;
      CUR_PATTERN := Operator_Symbol;
    when '-' =>
      CHAR_ADVANCE;
      CUR_PATTERN := Operator_Symbol;
      case CURRENT_CHAR is
        when '-' =>
          CHAR_ADVANCE;
          CUR_PATTERN := Comment;
          loop
            case CURRENT_CHAR is
              when ASCII.HT =>
                CHAR_ADVANCE;
              when ' '..'~' =>
                CHAR_ADVANCE;
              when others => exit;
            end case;
          end loop;
        when others => null;
```

```
    end case;
  when '(' =>
    CHAR_ADVANCE;
    CUR_PATTERN := Left_Parenthesis ;
  when 'A'..'Z' | 'a'..'z' =>
    CHAR_ADVANCE;
    CUR_PATTERN := Identifier;
    loop
      case CURRENT_CHAR is
        when '_' =>
          LOOK_AHEAD;
          case CURRENT_CHAR is
            when 'A'..'Z' | 'a'..'z' =>
              CHAR_ADVANCE;
            when '0'..'9' =>
              CHAR_ADVANCE;
            when others =>
              CUR_CHAR_NDX := FALL_BACK_NDX;
              LOOK_AHEAD_FAILED := TRUE;
          end case;
        when 'A'..'Z' | 'a'..'z' =>
          CHAR_ADVANCE;
        when '0'..'9' =>
          CHAR_ADVANCE;
        when others => exit;
      end case;
    exit when LOOK_AHEAD_FAILED;
    end loop;
  when '0'..'9' =>
    CHAR_ADVANCE;
    CUR_PATTERN := Decimal_Literal;
    case CURRENT_CHAR is
      when '.' | '0'..'9' | 'E' | '_' =>
        loop
          case CURRENT_CHAR is
            when '0'..'9' =>
              CHAR_ADVANCE;
            when '_' =>
              LOOK_AHEAD;
              case CURRENT_CHAR is
                when '0'..'9' =>
                  CHAR_ADVANCE;
                when others =>
                  CUR_CHAR_NDX := FALL_BACK_NDX;
                  LOOK_AHEAD_FAILED := TRUE;
              end case;
            when others => exit;
          end case;
        exit when LOOK_AHEAD_FAILED;
```

```
end loop;
if CUR_PATTERN /= UNRECOGNIZED then
  case CURRENT_CHAR is
    when '.' =>
      LOOK_AHEAD;
      case CURRENT_CHAR is
        when '0'..'9' =>
          CHAR_ADVANCE;
          if not LOOK_AHEAD_FAILED then
            case CURRENT_CHAR is
              when '0'..'9' | 'E' | '_' =>
                loop
                  case CURRENT_CHAR is
                    when '0'..'9' =>
                      CHAR_ADVANCE;
                    when '_' =>
                      LOOK_AHEAD;
                      case CURRENT_CHAR is
                        when '0'..'9' =>
                          CHAR_ADVANCE;
                        when others =>
                          CUR_CHAR_NDX := FALL_BACK_NDX;
                          LOOK_AHEAD_FAILED := TRUE;
                      end case;
                    when others => exit;
                  end case;
                exit when LOOK_AHEAD_FAILED;
                end loop;
                if not LOOK_AHEAD_FAILED then
                  case CURRENT_CHAR is
                    when 'E' =>
                      LOOK_AHEAD;
                      case CURRENT_CHAR is
                        when '+' | '-' =>
                          LOOK_AHEAD;
                          case CURRENT_CHAR is
                            when '0'..'9' =>
                              CHAR_ADVANCE;
                              if not LOOK_AHEAD_FAILED then
                                loop
                                  case CURRENT_CHAR is
                                    when '0'..'9' =>
                                      CHAR_ADVANCE;
                                    when '_' =>
                                      LOOK_AHEAD;
                                      case CURRENT_CHAR is
                                        when '0'..'9' =>
                                          CHAR_ADVANCE;
                                        when others =>
```

```
                                                CUR_CHAR_NDX := FALL_BACK_NDX;
                                                LOOK_AHEAD_FAILED := TRUE;
                                            end case;
                                          when others => exit;
                                        end case;
                                      exit when LOOK_AHEAD_FAILED;
                                      end loop;
                                    end if;
                                  when others =>
                                    CUR_CHAR_NDX := FALL_BACK_NDX;
                                    LOOK_AHEAD_FAILED := TRUE;
                                end case;
                              when '0'..'9' =>
                                CHAR_ADVANCE;
                                if not LOOK_AHEAD_FAILED then
                                  loop
                                    case CURRENT_CHAR is
                                      when '0'..'9' =>
                                        CHAR_ADVANCE;
                                      when '_' =>
                                        LOOK_AHEAD;
                                        case CURRENT_CHAR is
                                          when '0'..'9' =>
                                            CHAR_ADVANCE;
                                          when others =>
                                            CUR_CHAR_NDX := FALL_BACK_NDX;
                                            LOOK_AHEAD_FAILED := TRUE;
                                        end case;
                                      when others => exit;
                                    end case;
                                  exit when LOOK_AHEAD_FAILED;
                                  end loop;
                                end if;
                              when others =>
                                CUR_CHAR_NDX := FALL_BACK_NDX;
                                LOOK_AHEAD_FAILED := TRUE;
                            end case;
                          when others => null;
                        end case;
                      end if;
                    when others =>
                      CUR_CHAR_NDX := FALL_BACK_NDX;
                      LOOK_AHEAD_FAILED := TRUE;
                  end case;
                end if;
              when others =>
                CUR_CHAR_NDX := FALL_BACK_NDX;
                LOOK_AHEAD_FAILED := TRUE;
            end case;
```

```
when 'E' =>
  LOOK_AHEAD;
  case CURRENT_CHAR is
    when '+' | '-' =>
      LOOK_AHEAD;
      case CURRENT_CHAR is
        when '0'..'9' =>
          CHAR_ADVANCE;
          if not LOOK_AHEAD_FAILED then
            loop
              case CURRENT_CHAR is
                when '0'..'9' =>
                  CHAR_ADVANCE;
                when '_' =>
                  LOOK_AHEAD;
                  case CURRENT_CHAR is
                    when '0'..'9' =>
                      CHAR_ADVANCE;
                    when others =>
                      CUR_CHAR_NDX := FALL_BACK_NDX;
                      LOOK_AHEAD_FAILED := TRUE;
                  end case;
                when others => exit;
              end case;
            exit when LOOK_AHEAD_FAILED;
            end loop;
          end if;
        when others =>
          CUR_CHAR_NDX := FALL_BACK_NDX;
          LOOK_AHEAD_FAILED := TRUE;
      end case;
    when '0'..'9' =>
      CHAR_ADVANCE;
      if not LOOK_AHEAD_FAILED then
        loop
          case CURRENT_CHAR is
            when '0'..'9' =>
              CHAR_ADVANCE;
            when '_' =>
              LOOK_AHEAD;
              case CURRENT_CHAR is
                when '0'..'9' =>
                  CHAR_ADVANCE;
                when others =>
                  CUR_CHAR_NDX := FALL_BACK_NDX;
                  LOOK_AHEAD_FAILED := TRUE;
              end case;
            when others => exit;
          end case;
```

```
                          exit when LOOK_AHEAD_FAILED;
                          end loop;
                        end if;
                      when others =>
                        CUR_CHAR_NDX := FALL_BACK_NDX;
                        LOOK_AHEAD_FAILED := TRUE;
                    end case;
                  when others => null;
                end case;
              end if;
            when others =>
              CUR_PATTERN := UNRECOGNIZED;
          end case;
        when ' ' =>
          CHAR_ADVANCE;
          CUR_PATTERN := White_Space;
          loop
            case CURRENT_CHAR is
              when ' ' =>
                CHAR_ADVANCE;
              when others => exit;
            end case;
          end loop;
        when others =>
          CHAR_ADVANCE;
          CUR_PATTERN := UNRECOGNIZED;
      end case;
    end;

  end SAMPLE_TOKEN_STREAM;
```

APPENDIX C - SAMPLE LEXICAL ANALYZER TEST PROGRAM

This appendix contains a test program that calls the sample lexical analyzer discussed in Section 4. This program simply reports the token information returned by the analyzer. It includes definitions for the types TOKEN and TOKEN_TYPE, the GET_CHARACTER procedure, and the MAKE_TOKEN function required by the analyzer. It also includes the analyzer package specification and the "is separate" declaration for the analyzer package body produced by the generator.

```
with  INTEGER_TEXT_IO, TEXT_IO;

procedure SAMPLE_TEST_PROGRAM is

   --  This procedure is a sample test program for exercising code
   --  produced by the Lexical Analyzer Generator.

   use INTEGER_TEXT_IO, TEXT_IO;


   type TOKEN_TYPE is
           (IDENT, LF_PAREN, NOT_MINE, NUMBER, OPERATOR, RT_PAREN);

   subtype SHORT_STRING is STRING(1..12);

   type TOKEN is
      record
         KIND: TOKEN_TYPE;
         PRINT_VALUE: SHORT_STRING;
         LINE_NUMBER: INTEGER;
      end record;


   EOS: BOOLEAN;
   TOK: TOKEN;


   procedure GET_CHARACTER( EOS: out BOOLEAN;
                 NEXT: out CHARACTER;
                 MORE: in BOOLEAN := TRUE ) is
   --  Produce input characters for the lexical analyzer.
   begin
      if  END_OF_FILE(STANDARD_INPUT)  then
         EOS := TRUE;
      elsif  END_OF_LINE(STANDARD_INPUT)  then
         SKIP_LINE(STANDARD_INPUT);
         EOS := FALSE;
         NEXT := ASCII.CR;
      else
         EOS := FALSE;
         GET(STANDARD_INPUT,NEXT);
      end if;
   end;


   function  MAKE_TOKEN(KIND: TOKEN_TYPE; SYMBOL: STRING; LINE_NUMBER: NATURAL )
         return  TOKEN  is

   --  construct a token value from input lexical information
```

```
      function CVT_STRING( STR: in STRING ) return SHORT_STRING is
      --  Convert an arbitrary-length string to a fixed length string.
         RESULT: SHORT_STRING;
      begin
         for  I in SHORT_STRING'RANGE  loop
            if  I <= STR'LAST  then
               RESULT(I) := STR(I);
            else
               RESULT(I) := ' ';
            end if;
         end loop;
         return RESULT;
      end;

   begin
      return  TOKEN'(KIND, CVT_STRING(SYMBOL), LINE_NUMBER);
   end;


   package SAMPLE_TOKEN_STREAM is

     procedure ADVANCE(EOS: out BOOLEAN;
                       NEXT: out TOKEN;
                       MORE: in BOOLEAN := TRUE);

   end SAMPLE_TOKEN_STREAM;

   package body SAMPLE_TOKEN_STREAM is separate;


begin

   loop
      SAMPLE_TOKEN_STREAM.ADVANCE(EOS,TOK);
   exit when EOS;
      PUT(TOK.PRINT_VALUE);
      PUT("  ");
      PUT(TOK.LINE_NUMBER);
      PUT("  ");
      case TOK.KIND is
         when IDENT     => PUT("Identifier");
         when LF_PAREN  => PUT("Left Parenthesis");
         when NOT_MINE  => PUT("Unrecognized");
         when NUMBER    => PUT("Number");
         when OPERATOR  => PUT("Operator");
         when RT_PAREN  => PUT("Right Parenthesis");
      end case;
      NEW_LINE;
```

```
   end loop;

end SAMPLE_TEST_PROGRAM;
```

APPENDIX D - SAMPLE LEXICAL ANALYZER TEST DATA

INPUT:

```
+  -  *  /   -- the operators
123 45.67 89E10  -- numbers
ABC ijk XYZ  -- identifiers
-- This is a comment

-- ^ A blank line
ABC + (123 - xyz) * 0.456  -- one expression
123.456E78 / -123.456E-78  -- another
1.0 - sin(theta)  -- and another
```

OUTPUT:

```
+                      1  Operator
-                      1  Operator
*                      1  Operator
/                      1  Operator
123                    2  Number
45.67                  2  Number
89E10                  2  Number
ABC                    3  Identifier
ijk                    3  Identifier
XYZ                    3  Identifier
ABC                    7  Identifier
+                      7  Operator
(                      7  Left Parenthesis
123                    7  Number
-                      7  Operator
xyz                    7  Identifier
)                      7  Right Parenthesis
*                      7  Operator
0.456                  7  Number
123.456E78             8  Number
/                      8  Operator
-                      8  Operator
123.456E-78            8  Number
1.0                    9  Number
-                      9  Operator
sin                    9  Identifier
(                      9  Left Parenthesis
theta                  9  Identifier
)                      9  Right Parenthesis
```

Distribution List for IDA Paper P-2109

NAME AND ADDRESS	NUMBER OF COPIES
Sponsor	
Dr. John F. Kramer Program Manager STARS DARPA/ISTO 1400 Wilson Blvd. Arlington, VA 22209-2308	1
Other	
Defense Technical Information Center Cameron Station Alexandria, VA 22314	2
IDA	
General W.Y. Smith, HQ	1
Ms. Ruth L. Greenstein, HQ	1
Mr. Philip L. Major, HQ	1
Dr. Robert E. Roberts, HQ	1
Dr. Richard J. Ivanetich, CSED	1
Ms. Katydean Price, CSED	1
Dr. Reginald N. Meeson, CSED	1
IDA Control & Distribution Vault	2